Oxford

STOCK No. RK1556

Dubl-Deep Tab Pat. No. 2,037,579

Exclusive "Padded Shoulder" Feature

MADE IN U.S.A.

COO-3077-64

Systems Programming Languages and Graphics Terminals[1]

Thomas Stuart
Courant Institute of Mathematical Sciences
New York University

Abstract

Faced with the design of an interactive graphics system, many researchers with widely differing applications have settled on a random access CRT display attached to a miniprocessor. In most cases this small computer is in turn linked to a larger time-shared facility, and it is the software implications of this overall hardware design with which this paper is concerned.

Such a setup is usually arrived at by a compromise between an estimate of required computational power and the reality of financial resources. The insertion of the small computer between the display device and the applications oriented large computer causes difficulties in three main areas:

(1) Decreased utility services, compilers, and other system software where it's needed.

(2) A more complex design for the graphics software, due both to an added communications channel and the division of labor between computers.

(3) A degradation of response time when interaction between display and application programs is necessary.

It is asserted here that the use of a systems programming language will aid in tackling each of these problems. With respect to the first, system software, one has an obvious application of the earliest arguments for a systems language. The extension of these arguments to the second difficulty, the design and implementation of a graphics system, receives major attention. Though a choice of programming languages would usually have no clear, direct effect on the third problem, response time, it does have an indirect influence since the response can never be completely independent of communications software. The disadvantages of a systems language are not to be ignored either. Finally, the use of a specific language, LITTLE, is described briefly.

Introduction

As with any new item of hardware, the development of graphics terminals brought a host of problems. Among the various types of terminals, one that has found a wider audience is the programmed beam CRT driven by a computer of relatively limited resources. Usually these resources are too limited for a complete resolution of the specific applications problem and, hence, a more versatile computer is used to handle the larger computational tasks.

At the outset, the advantages of such a design were not readily apparent.[2] Given a number of applications with varying graphical demands and varying computational requirements it was difficult for designers to choose an optimal system even when the system was to be used for a single application and its needs were well defined beforehand. The earliest solutions took one of two forms: if the computational tasks were large, plug the CRT into the large computer; if not, get a mini to drive the display. Neither solution was very practical with most problems.

When the memory and I/O facilities of a large computer are employed to drive the display directly the volume of data transmitted to maintain the image represented a serious drain on, and waste of, the resources available, degrading production in a time sharing system. On the other hand, if the driver is a more or less independent processor with a small memory set by a program in the main memory only as needed, the degradation this time occurs at the terminal, which has real time requirements undreamt of at IBM, CDC, or elsewhere. So long as an image is just being regenerated, the main processor need contribute little time or space to graphics, but whenever the picture changes in some manner not handled by the hardware of the CRT controller, then, within milliseconds, a generally large program must be in central memory and executing. The percentage of the time which must be devoted to this program of course depends on the application, but many problems will vary during the course of a job from a fraction of 1 per cent all the way up to 100 per cent. No time-sharing system which allowed this would be time-shared any longer. Essentially, the only types of interactive graphic display that can profitably be driven by a time-shared computer (with or without an intermediary processor) are displays of a relatively static nature,

alphanumeric displays, storage tubes, point plotters, etc., and we are not concerned with their special problems here. Whether a design lacking degradation on either end might be implemented is problematical, for such a facility would be a costly feature.

The other solution, driving the display from a stand alone minicomputer, was simply inadequate for the computational necessities of most problems, if not initially, then as additional projects were attempted. Expanding applications can outstrip attempts to upgrade performance of the minicomputer with great speed. However, this approach did possess the advantage of adaptability in some cases, for, if a large computer was near enough for a low cost, high volume transmission line, then the computational portion of the graphics problem could be undertaken, and all one had to face was reprogramming the system to reflect the minicomputer's shifted status - from central processor to intelligent terminal. The choice of this latter system, while it may in some cases involve substantial effort to convert, was eventually advantageous for its users, for the choice of a larger computer as driver has led to, or will lead to, the junking of both software and hardware.

The decision to employ this type of interactive graphics terminal - CRT attached to minicomputer attached to main computer - is not an easy one, since response time between applications program and display is degraded, and since it can only be more difficult to develop software, but it is a choice that experience forces, because there is no avoiding the need for the computational power of a large computer, and there are no projects that justify a dedicated mammoth and the concomitant waste. Once that basic choice is made there are still many questions to be answered concerning hardware characteristics of the display, the minicomputer, and the peripherals. Answers are impossible to generalize and will often depend more directly upon the specific application and financial backing. In any case, though these considerations are very important, the remainder of this paper is concerned with the other half of a system design, software.

Software

Software difficulties increase enormously with this terminal design. First, and most obvious, it is because there are two computers instead of one to program. Communication between them is always a non-trivial problem, especially when entering a time-shared community with some real time desires. The software interface between the graphics routines on the mini and graphic processing in the main computer is only one of the added interfaces. There are now application routines both on the time-shared computer and at the terminal, each requiring interfaces with the graphic routines where one sufficed earlier.

Perhaps not so obvious is another effect of the two computer choice, an immediate repercussion being that every time a new routine is written there is a choice to be made: where will it run? The answer to this question is in many cases neither apparent nor trivial. Factors influencing the choice will include -

(1) The memory available at the terminal, both in core and in storage devices,
(2) The access time of these devices,
(3) The speed of the link (or links) to the main computer,
(4) The priority this particular task would be accorded in the main computer at this time,
(5) The frequency of performance of the task for each application, and
(6) The volume of data to be processed during a particular job.

Whereas the first three of these factors tend not to change frequently and do absolutely determine the location of many routines, the latter three are time dependent variables and merit more attention. Of course, we always wish to carry out a process on the minicomputer when possible. Whether or not it is feasible depends not only on the hardware available, but the run-time environment. The fourth, fifth and sixth factors constitute that environment.

The priority of a task coupled with the time of day can sometimes determine the optimum location for executing a routine. Especially would this be true for processes requiring a few seconds on a mini and an order of magnitude less in the time-shared facility. In peak use hours, the mini is still a preferable location, whereas at night it never would be.

The fifth factor, frequency of task performance, becomes important as a determinant only when high. For an infrequently performed task the location is relatively unimportant when a choice is available. When the speed considerations point to the main computer as the prime site, then it still may be desirable to have a back-up routine for avoiding down time impediments. But when the frequency is high, speed assumes large importance and we have a boosting of the importance of the priority factor just cited.

The last effect, the volume of data, is the most unpredictable in its importance. It is here that many graphics applications will envelop a range of specific jobs that include some production runs requiring a large computer and some that do not. An ad hoc decision is usually made to program for the worst case, a decision which is often forced by the difficulty of reprogramming.

Specific cases which will be affected by one or more of these factors are, it should be emphasized, quite dependent on the hardware. That is, the affected routines will change from one system to another, but so long as a graphics project embraces a large number of tasks, there will be many falling in the range of interest.

Now that some of the complexities of the programming have been established, it may be reasonable to summarize, to characterize the problem.

First, the problem is many faceted. It embraces transformation and manipulation routines, access of various utility functions on both computers, an executive, probably some data analysis routines, more than likely a complex data structure.

Second, the program requires a formal structure for interfacing its divisions and also because it is to be written by more than one person.

Third, there is a high emphasis on the speed of execution, both because there are some real time criteria, and because it is designed for continuous or frequent production.

Fourth, it is likely to change frequently with changing needs, priorities, applications, and hardware.

Fifth, since it not only serves as a service program for large applications, but also carries out simple display problems as a stand along package, it must be supremely easy to use.

Sixth, some of the routines or tasks are to be written for two machines to best advantage.

In many interactive graphics projects a seventh characteristic would also be important, hardware independence. This takes either of two forms, an extension of the sixth characteristic to the whole of the graphics program because the package is to execute on two, different minicomputers, or, alternatively, some routines are to produce images on two, different display devices.

Then, it is appropriate to ask, what is the vehicle through which this problem is to be implemented, how shall it be programmed, in other words, which language?

Language

The above summary of the problem is a re-statement in specific graphics terms of the definition of a system problem given in the recent review article, "Systems Programming Languages".[3] Obviously, then, we have a system problem for which systems programming languages were developed. Obviously, then, we should use a systems language. Or perhaps not.

There are several very good reasons why a systems language should not be used. First, it doesn't exist. Though this is not likely the case in most large, time-shared systems, it is very frequently the case for many minicomputers. If it doesn't exist, then its creation is generally beyond the resources of a graphics project. Another reason is that it just isn't fast enough. Even though systems languages are designed for efficiency, they simply cannot match a routine coded in an assembly language. A systems language is also often machine dependent where there may be strong reasons to lean in the opposite direction, writing in FORTRAN, for instance. Then, too, FORTRAN has an audience in the applications field where there may well be need to interact with

and modify the graphics program itself. There are many other reasons for the persistence of FORTRAN in the graphics field, but as Newman and Sproull[4] state in an introduction to a discussion of the language in their text, Principles of Interactive Computer Graphics, "one language which has been used time and again as a graphics system basis is FORTRAN. If it were not for this, there would be little need to mention FORTRAN at all, for its performance in meeting our criteria is abysmal". The discussion that follows the quote is recommended reading.

My own list of most negative attributes in its use in graphics include slow speed, poor character handling, no macros, no bit or field operators, and an extremely poor base upon which to build a graphic language. None the less, being dissatisfied with FORTRAN is still insufficient incentive if no alternative is readily available, and some of the deficiencies pointed out in the aforementioned review[2] are serious. Some systems languages are no faster than FORTRAN; some are quite complicated to learn; most are very machine dependent. It is understandable that many will rely on the assembly language for the most crucial speed, bit and field needs, and just put up with FORTRAN's other attributes.

However, I was luckier. At the Courant Institute, Jacob Schwartz began development of a language called LITTLE in 1968.[5] The two basic goals of this language are machine independence and efficiency.[6] The LITTLE compiler aids the first goal by expressing programs as 'machine' language for an 'unknown machine', unknown, that is, at the time of program construction, with characteristics such as the word size considered a compile time parameter. The machine dependent section of LITTLE is a routine that maps the 'unknown machine' language onto the real machine, but this task is not at all extraordinary for any new machine, because the LITTLE compiler is written entirely in LITTLE. The approach bears some similarity to the 'abstract' machine of the STAGE2 system,[7] with the practical difference of less independence and more efficiency. Thus far, the second goal, efficiency, has only been measured on the CDC 6600 and is excellent. Efficiency results on the IBM 360 are expected shortly.

The structure of LITTLE includes MACROS, the usual arithmetic, logical and relational operations, bit and field operators, a single data type - the bit string,

the common IF...THEN...ELSE, DO, UNTIL, and WHILE constructions, conditional and unconditional transfers, local and global variables and a general program structure reminiscent of FORTRAN. In essence, if one took FORTRAN and added most of the options a systems programmer would want, one would have LITTLE.

From my own standpoint this structure has the additional advantages of making a link to existing FORTRAN routines easy, encouraging easy use to present FORTRAN-only programmers, and providing a vastly improved base for a graphics language.

Currently it is not a trivial exercise to write a machine block, with an estimate of four months of intensive work for a Honeywell Series 16 minicomputer, but this will decrease with the future expression of the 'unknown machine' in a formal language.

Acknowledgement

Jacob Schwartz, David Shields, and Edith Deak were all very patient and very helpful in introducing me to LITTLE. Elias Guth aided on the other end, getting the first code for the Series 16 machines out.

References

1. This work has been supported by PHS grant number NS-10072. General support by the AEC Computing and Applied Math Center under contract AT(11-1)3077 is also gratefully acknowledged.

2. Foley, J.D., "An Approach to the Optimum Design of Computer Graphics Systems", Comm. ACM, 14, 380 (1971).

3. Bergeron, R.D., Gannon, J.D., Schecter, D.P., Tompa, F.W., and Van Dam, A., "Systems Programming Languages" in Advances in Computers, 12, Academic, New York (1972), p. 177.

4. Newman, W.M. and Sproull, R.F., Principles of Interactive Computer Graphics, McGraw-Hill, New York (1973) p. 362.

5. Cocke, J. and Schwartz, J., Programming Languages and their Compilers, Courant Institute of Mathematical Sciences, New York University (1970).

6. Shields, D., "Guide to the LITTLE Language", LITTLE Newsletter No. 33, Courant Institute, New York University (1974).

7. Poole, P.C. and Waite, W.M., "Portability and Adaptability" in Advanced Course on Software Engineering, Springer-Verlag, Berlin (1973).

book may be kept

FOURTEEN DAYS

e will be charged for each day the book is kept overtime.

COO-3077-64 c.2

Stuart

Systems programming languages and graphics terminals.

COO-3077-64 c.2

Stuart

Systems programming languages and graphics terminals.

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012